ELEC6021 Research Methods assignment: Matlab

This assignment forms your assessment for the "Introduction to Matlab" part of ELEC6021 Research Methods and contributes 25% of your mark for that course. You are required to work entirely on your own and produce a write-up, which only needs to include the Matlab code, flow charts, results and discussions that are directly requested in the **bold** paragraphs below. Since this assignment is worth five credits, it should take you up to 50 hours to complete it. Please do not spend any more time than this since I'm sure you have better things to be doing! You need to submit your write-up electronically at C-BASS

`https://handin.ecs.soton.ac.uk/handin/0910/ELEC6021/1/`

before 4pm on Thursday 12/11/2009. You also need to print out your write-up, staple or bind it together with your C-BASS receipt and submit it at the ECS reception before the same deadline.

Here is some advice for completing this assignment:

- The three exercises are in no particular order. You may find it easier to complete the exercises in an order other than 1, 2, 3. For this reason, you should read through all of this document before deciding which exercise to tackle first.
- If you get stuck on a particular exercise, you should leave it for a while and move on to another one. This is a better use of your time and you might learn something in the new exercise that helps you to get unstuck in the other one. In the event that you run out of time, it would be better to have done quite well in all exercises rather than really well in one, but badly in the others.
- Try to make your Matlab code as reusable, elegant, efficient, structured and human readable as possible because marks are awarded for this. You should include useful, concise and relevant comments in your code to help explain

its non-trivial features. The variable names you choose should either correspond to the ones used in this document or should be descriptive, concise and relevant. You should also include adequate error checking in your Matlab functions to ensure that their input arguments are valid.

- Make sure that your flow charts and results clearly show what they are intended to. For example, remember to annotate the axes of your plots.
- More advice is offered for each of the individual exercises below.
- Have fun!

Rob Maunder
rm@ecs.soton.ac.uk

Exercise 1

The values of N unknown variables $x_1, x_2, x_3, \ldots, x_N$ can be obtained by solving the following set of N *simultaneous equations*, in which all other values are known constants.

$$\begin{array}{ccccccccccc}
a_{11}x_1 & + & a_{12}x_2 & + & a_{13}x_3 & + & \cdots & + & a_{1N}x_N & = & b_1 \\
a_{21}x_1 & + & a_{22}x_2 & + & a_{23}x_3 & + & \cdots & + & a_{2N}x_N & = & b_2 \\
a_{31}x_1 & + & a_{32}x_2 & + & a_{33}x_3 & + & \cdots & + & a_{3N}x_N & = & b_3 \\
\vdots & & \vdots & & \vdots & & \ddots & & \vdots & & \vdots \\
a_{N1}x_1 & + & a_{N2}x_2 & + & a_{N3}x_3 & + & \cdots & + & a_{NN}x_N & = & b_N
\end{array}$$

These equations may be written in matrix form $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ as follows.

$$\underbrace{\begin{bmatrix}
a_{11} & a_{12} & a_{13} & \cdots & a_{1N} \\
a_{21} & a_{22} & a_{23} & \cdots & a_{2N} \\
a_{31} & a_{32} & a_{33} & \cdots & a_{3N} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
a_{N1} & a_{N2} & a_{N3} & \cdots & a_{NN}
\end{bmatrix}}_{\mathbf{A}} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} b_1 \\ b_2 \\ b_3 \\ \vdots \\ b_N \end{bmatrix}}_{\mathbf{b}}$$

Matlab can easily solve these simultaneous equations by calculating $\mathbf{x} = \mathbf{A}^{-1} \cdot \mathbf{b}$. However, in this exercise, you won't have this luxury! Instead, you need to write a Matlab function that will transform $\mathbf{A}$ and $\mathbf{b}$ into a form that makes solving the simultaneous equations more manageable.

There are certain transformations that will maintain the relationship $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ when they are simultaneously applied to *both* $\mathbf{A}$ and $\mathbf{b}$. These are

- multiplying a particular row of $\mathbf{A}$ and $\mathbf{b}$ by a constant,
- adding a row of $\mathbf{A}$ and $\mathbf{b}$ to any other and
- swapping any two rows of $\mathbf{A}$ and $\mathbf{b}$.

Using these transformations, $\mathbf{A}$ and $\mathbf{b}$ can be manipulated into *upper triangular form*, becoming $\mathbf{A}'$ and $\mathbf{b}'$, respectively. Here, $\mathbf{A}' \cdot \mathbf{x} = \mathbf{b}'$ and all elements below the diagonal of $\mathbf{A}'$ are equal to zero.

$$\underbrace{\begin{bmatrix}
a'_{11} & a'_{12} & a'_{13} & \cdots & a'_{1N} \\
0 & a'_{22} & a'_{23} & \cdots & a'_{2N} \\
0 & 0 & a'_{33} & \cdots & a'_{3N} \\
\vdots & \vdots & \vdots & \ddots & \vdots \\
0 & 0 & 0 & \cdots & a'_{NN}
\end{bmatrix}}_{\mathbf{A}'} \cdot \underbrace{\begin{bmatrix} x_1 \\ x_2 \\ x_3 \\ \vdots \\ x_N \end{bmatrix}}_{\mathbf{x}} = \underbrace{\begin{bmatrix} b'_1 \\ b'_2 \\ b'_3 \\ \vdots \\ b'_N \end{bmatrix}}_{\mathbf{b}'}$$

Consider the following example.

$$\begin{bmatrix} 3 & 1 \\ 1 & 2 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ 5 \end{bmatrix}$$

We can multiply the second row by -3 to obtain the following.

$$\begin{bmatrix} 3 & 1 \\ -3 & -6 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ -15 \end{bmatrix}$$

The following is obtained when we add the first row to the second row.

$$\begin{bmatrix} 3 & 1 \\ 0 & -5 \end{bmatrix} \cdot \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} = \begin{bmatrix} 5 \\ -10 \end{bmatrix}$$

The equation is now in upper triangular form, making it easy to solve the simultaneous equations and obtain the values of x_1 and x_2. More specifically, the second row tells us that $-5x_2 = -10$, giving $x_2 = 2$. The first row reveals that $3x_1 + x_2 = 5$. Substituting for x_2 and solving yields $x_1 = 1$. Of course, more steps are required when N is greater and that's where your Matlab skills come in to play!

Draw a *flow chart* and write some Matlab code for a function that obtains the upper triangular form by manipulating $\mathbf{A}$ and $\mathbf{b}$ into $\mathbf{A}'$ and $\mathbf{b}'$, respectively. Your function must also obtain the solution vector $\mathbf{x}$ for the simultaneous equations. Therefore, the first line of your Matlab function is required to be as follows.

```
function [x, Aprime, bprime] = solve(A, b)
```

In cases where no errors occur, `x` and `bprime` are required to have the same dimensions as `b`, while `Aprime` is required to have the same dimensions as `A`.

Try your function with the following six tests (some of which have particular features that will challenge your function's error checking).

1. $\mathbf{A} = \begin{bmatrix} 1 & 1 & 0 \\ 2 & 1 & 1 \\ 1 & 2 & 3 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 3 \\ 7 \\ 14 \end{bmatrix}.$

2. $\mathbf{A} = \begin{bmatrix} 0 & 1 & 3 & 2 \\ 2 & 1 & 4 & 0 \\ 1 & 1 & 2 & 1 \\ 1 & 2 & 3 & 1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 14 \\ 11 \\ 11 \\ 15 \end{bmatrix}.$

3. $\mathbf{A} = \begin{bmatrix} 1 & 1 & 3 \\ 1 & 1 & 1 \\ 2 & 1 & 1 \end{bmatrix}, \mathbf{b} = \begin{bmatrix} 4 \\ 2 \\ 3 \end{bmatrix}.$

4. $\mathbf{A} = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 2 \\ 3 \end{bmatrix}$.

5. $\mathbf{A} = \begin{bmatrix} 1 & 2 & 0 \\ 2 & 0 & 1 \\ 0 & -4 & 1 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 6 \\ 7 \\ -5 \end{bmatrix}$.

6. $\mathbf{A} = \begin{bmatrix} 3 & 2 & 0 & 1 \\ 2 & 1 & 3 & 2 \\ 4 & 0 & 1 & 1 \\ 5 & -2 & 2 & 1 \end{bmatrix}$, $\mathbf{b} = \begin{bmatrix} 11 \\ 21 \\ 11 \\ 11 \end{bmatrix}$.

Include your flow chart, a listing of your function and its responses to the above tests in your write-up. Also, explain the particular features exhibited by the tests that generate errors.

Here is some advice for completing this exercise:

- Begin by working out how to perform each of the three transformations listed above in Matlab. The colon operator will be useful here.
- The six tests listed above are ordered according to how difficult it is to get your function to respond correctly. Concentrate on getting your function to give the correct response to the first test, before considering the second and so on. Try solving the tests by hand so that you can see what your function is up against!
- Draw your flow chart on a computer so that it is easy to move parts around and make changes to it. You should be doing this in parallel with writing your Matlab code, rather than leaving the flow chart to the end. You should find that doing things this way will make it easier for you to break the problem down.
- Figure 1 provides a flow chart that will help you get started. Note that this flow chart does not obtain the solution vector, it won't work for all of the tests listed above and it doesn't include any error checking. Your final flow chart will therefore have to be more sophisticated than this one.
- Not including argument checking and comments, the Matlab function can be written using only 25 lines of code, without taking any shortcuts to cut this down. Also, a final flow chart comprising just 20 items is possible. If you find that your function and flow chart are much bigger than these, then you are probably overcomplicating things.

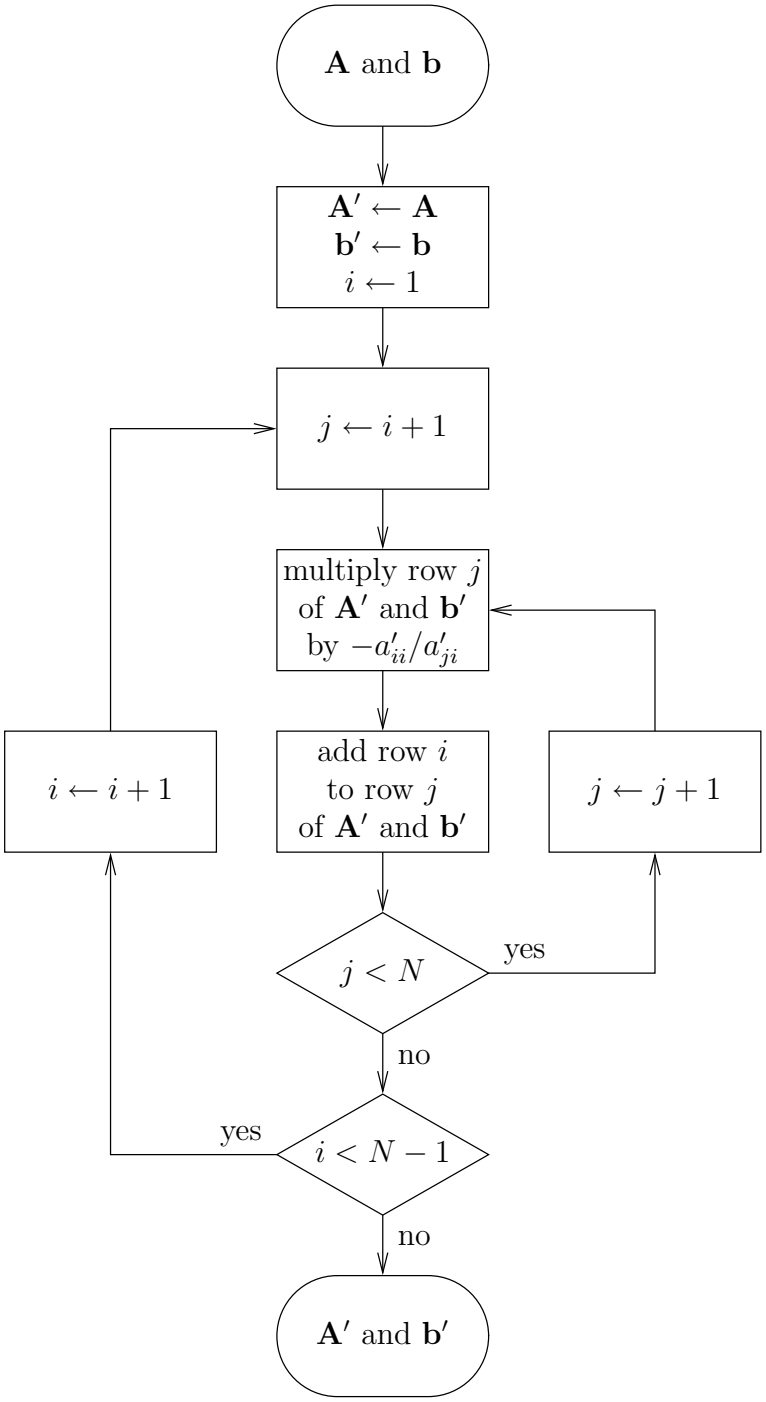

Figure 1: Flow chart for rearranging an N by N matrix $\mathbf{A}$ and an N by 1 vector $\mathbf{b}$ into upper triangular form, $\mathbf{A}'$ and $\mathbf{b}'$, respectively. Here, a'_{ij} is the element of the matrix $\mathbf{A}'$ in row i and column j.

- Simultaneous equations have many diverse applications, including circuit analysis, economics and optimisation.

Exercise 2

The mathematical function $y = f(\mathbf{x})$ has one dependent variable y, but N independent variables, which are provided in the vector $\mathbf{x} = [x_1, x_2, x_3, \ldots, x_N]$. Suppose that you want to find the particular values of the independent variables $\mathbf{x}$ that maximise the value of the dependent variable y. However, consider the case where the function is like a 'black box'; you can input some particular values of the independent variables $\mathbf{x}$ and observe the resultant value of the dependent variable y, but you can't see how the function is *defined*. Since you don't know how the value of y depends on those of $\mathbf{x}$, you are prevented from using differentiation techniques to find the particular values of $\mathbf{x}$ that maximise the value of y. In this case, the *steepest ascent optimisation algorithm* comes to the rescue!

The steepest ascent optimisation algorithm is an iterative algorithm for finding the particular values of $\mathbf{x}$ that maximise the value of $y = f(\mathbf{x})$. This algorithm starts with a 1 by N vector of initial values for the independent variables $\mathbf{x}_0$. It then assesses the *gradient* of the function $f(\mathbf{x})$ in the vicinity of these values $\mathbf{x}_0$. Note that this gradient $\mathbf{g}_0 = \nabla f(\mathbf{x}_0)$ has direction has well as magnitude and so $\mathbf{g}_0$ is also a 1 by N vector, just like $\mathbf{x}_0$. The direction of the gradient $\mathbf{g}_0$ points 'uphill', towards better values of the independent variables $\mathbf{x}$, which give higher values for $y = f(\mathbf{x})$. This allows us to easily replace our initial independent variable values $\mathbf{x}_0$ with new better ones $\mathbf{x}_1$. More specifically, we can obtain $\mathbf{x}_1$ according to $\mathbf{x}_i = \mathbf{x}_{i-1} + \delta \cdot \mathbf{g}_{i-1}$, where δ is a small positive constant. Furthermore, we can iteratively repeat this process to improve upon $\mathbf{x}_1$ and so on. This iterative process can be terminated when $f(\mathbf{x}_i) - f(\mathbf{x}_{i-1}) < t$, where t is another small positive threshold constant.

The steepest ascent optimisation algorithm is exemplified in Figure 2, in which a surface plot is provided for a function $y = f_1(\mathbf{x})$ of two independent variables $\mathbf{x} = [x_1, x_2]$. In this example, the algorithm used initial independent variable values of $\mathbf{x}_0 = [0.5, -0.9]$ and it ran for 25 iterations. The evolution of these iterations is indicated using a series of linked dots, which plots $y_i = f_1(\mathbf{x}_i)$ against $\mathbf{x}_i$ for $i = 0, 1, 2, \ldots, 25$. As shown in Figure 2 the algorithm always progresses 'uphill', in the direction of the gradient $\mathbf{g}_i$. Furthermore, the algorithm progresses at a rate which is proportional to the magnitude of the gradient $\mathbf{g}_i$, by a coefficient equal to $\delta = 0.25$. As a result, big hops are used when the gradient is steep, but small hops are used when approaching the highest point of the surface plot. In this way, the steepest ascent optimisation algorithm is able to quickly and accurately determine the values of $\mathbf{x}$ that maximises the function $y = f_1(\mathbf{x})$.

In this exercise, you need to use Matlab to perform steepest ascent optimisation

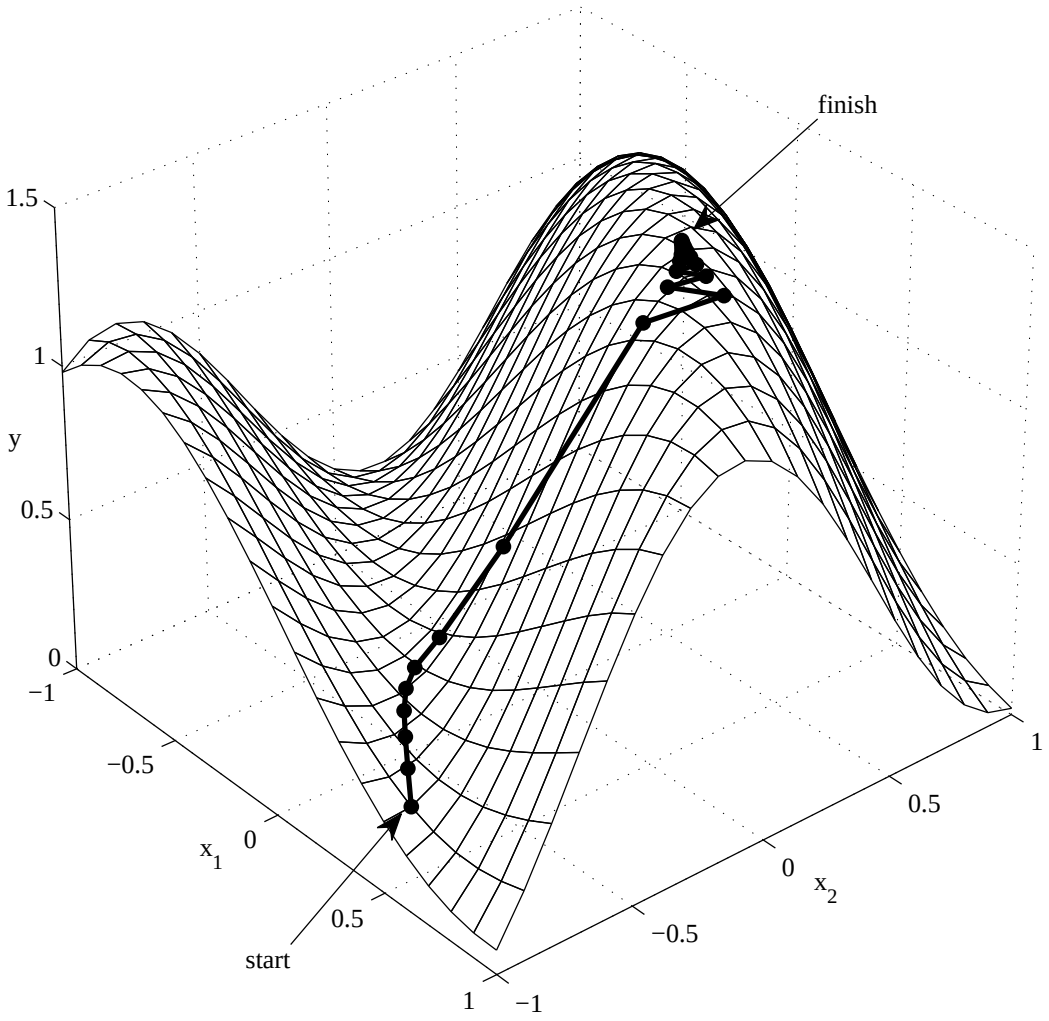

Figure 2: Example of using the steepest ascent optimisation algorithm to maximise $y = f_1(\mathbf{x})$. Here, $\mathbf{x} = [x_1, x_2]$, $\mathbf{x}_0 = [0.5, -0.9]$ and $\delta = 0.25$.

and find the independent variable values that maximise the mathematical functions that are listed below. You will need to write a Matlab function for each of these mathematical functions. Each of these Matlab functions is required to have a first line like the following example, which should be used for the first of the mathematical functions listed below, $y = f_1(\mathbf{x})$.

```
function y = f_1(x)
```

Here, the argument `x` is required to be a 1 by N vector that provides the independent variables $\mathbf{x} = [x_1, x_2, x_3, \ldots, x_N]$. Furthermore, `y` is required to be a scalar that returns the corresponding value of $y = f_1(\mathbf{x})$.

You will also need to draw a flow chart and write a Matlab function for performing the steepest ascent optimisation algorithm. Note that your *single* Matlab function is required to work for *all three* of the mathematical functions listed below. The first line of your Matlab function is required to be as follows.

```
function [X, y] = optimise(function_name, x_0, delta)
```

Here, the argument `function_name` is a function handle that allows you to specify which of the mathematical functions listed below to optimise. More specifically, this argument provides a string containing the name your corresponding Matlab function. For example, this string would have a value of `'f_1'`, if the mathematical function $y = f_1(\mathbf{x})$ was to be optimised. The argument `x_0` lets you specify the vector of initial independent variable values $\mathbf{x}_0$, as described above. The third argument `delta` is used to provide a value for the small positive scalar δ, which controls how far the steepest ascent algorithm will hop in each iteration. Your `optimise` function is required to have two return variables. The first `X` should be an M by N matrix, where M is the number of iterations that the steepest ascent algorithm runs for. Each row of the matrix `X` should provide the vector of independent variable values $\mathbf{x}_i$ that was used in a different iteration. More specifically, $\mathbf{x}_0$ should provide the first row of `X`, $\mathbf{x}_1$ should provide the second row and so on. The second return variable `y` should be an M by 1 vector that provides the corresponding dependent variable values y_i, where the first element provides y_0, the second provides y_1 and so on.

Write Matlab functions for each of the following three mathematical functions and use them as inputs to your `optimise` function.

1. $y = f_1(\mathbf{x})$,
   $\mathbf{x} = [x_1, x_2]$,
   $\mathbf{x}_0 = [0.5, -0.9]$,
   $f_1(x_1, x_2) = 1 - x_1^2/4 - x_2^2/4 - \sin(-2x_1 - 3x_2)/2$,
   $\delta = 0.25$.

2. $y = f_2(\mathbf{x})$,
   $\mathbf{x} = [x_1]$,
   $\mathbf{x}_0 = [220]$,
   $f_2(x_1) = 9.87\sin(2b(x_1)) - 7.53\cos(b(x_1)) - 1.5\sin(b(x_1))$,
   $b(x_1) = 2\pi(x_1 - 81)/365$,
   $\delta = 50$.

3. $y = f_3(\mathbf{x})$,
   $\mathbf{x} = [x_1, x_2, x_3]$,
   $\mathbf{x}_0 = [0, 0, 0]$,
   $f_3(x_1, x_2, x_3) = 9x_1 + 8x_2 + 7x_3 - 6x_1^2 - 5x_2^2 - 4x_3^2 + 3x_1x_2 + 2x_1x_3 + x_2x_3$,
   $\delta = 0.1$.

Include listings of your `f_1`, `f_2` and `f_3` Matlab functions in your write-up. Also include your flow chart and a listing of your `optimise` function. Take the results you obtained by passing your `f_1` function to your `optimise` function and use Matlab to draw a 3D figure similar to Figure 2. Similarly, draw an equivalent 2D plot for the results obtained using your `f_2` function. In addition to these plots, your write-up should include the matrix `X` and the vector `y` that you obtained by passing your `f_3` function to your `optimise` function.

Here is some advice for completing this exercise:

- Begin by writing the Matlab functions `f_1`, `f_2` and `f_3`.
- Figure 3 provides a flow chart that will help you to get started with the `optimise` function. This flow chart illustrates the process for determining the gradient of a function in the vicinity of particular values of its independent variables. When drawing your flow chart of the `optimise` function, you can use a single box to represent the process illustrated in Figure 3.
- Draw your flow chart of the `optimise` function on a computer so that it is easy to move parts around and make changes to it. You should be doing this in parallel with writing your Matlab code, rather than leaving the flow chart to the end. You should find that doing things this way will make it easier for you to break the problem down.
- I found that values of $\epsilon = 0.0001$ and $t = 0.0001$ work well.
- You will probably find the built-in Matlab functions `feval`, `plot`, `plot3`, `meshgrid`, `surf` and `hold` to be useful in this exercise.

- Not including argument checking and comments, the `optimise` function can be written using only 23 lines of code, without taking any shortcuts to cut this down. Also, a flow chart comprising just 7 items is possible, if a single box is used to represent the process illustrated in Figure 3. If you find that your function and flow chart are much bigger than these, then you are probably overcomplicating things.

- You may be interested to find out that the function $y = f_2(\mathbf{x})$ listed above is the *equation of time*. Look it up you want to find out more.

- Optimisation algorithms have many diverse applications, including trade theory, design problems and economic decision making.

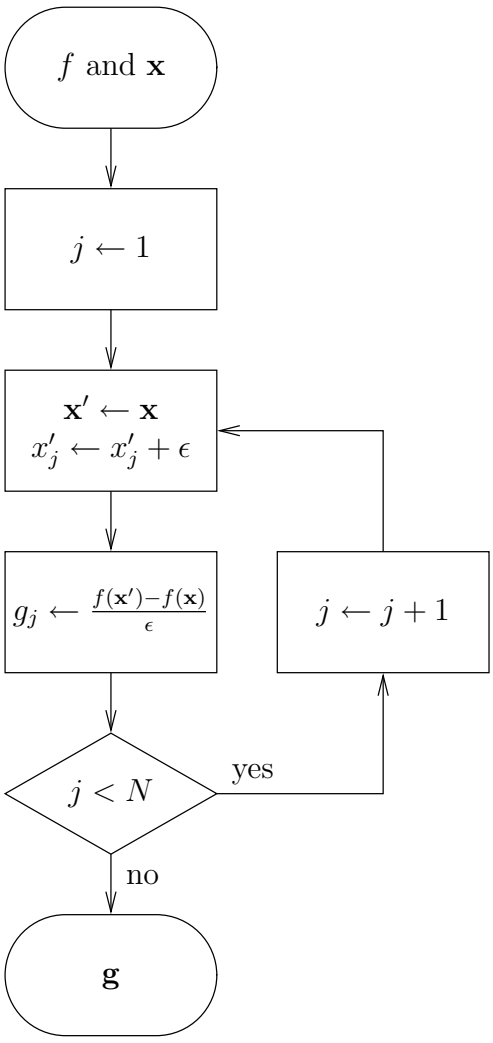

Figure 3: Flow chart for finding the gradient $\mathbf{g}$ of the function f in the vicinity of the particular values provided for the N independent variables $\mathbf{x} = [x_1, x_2, x_3, \ldots, x_N]$. Here, g_j is the j^{th} element in the 1 by N vector $\mathbf{g}$ and ϵ is a small positive constant.

Exercise 3

Consider a game where a player rolls some dice[1]. If more than y number of the dice have the same value then the player gets a score of zero. Otherwise, the player gets a score equal to the sum of the dice values. In this game, the player gets to choose the number x of dice to roll; the more dice the player rolls, the higher the sum of their values is likely to be, but the higher the chance of getting more than y dice with the same value. For example, in the case where $y = 2$, the $x = 4$ dice ⚂⚅⚀⚅ would get a score of 12, since no more than $y = 2$ of the dice have the same value. By contrast, the $x = 6$ dice ⚃⚂⚂⚂⚂⚂ would get a score of zero, since the number of dice having a value ⚂ is greater than $y = 2$.

For each pair of $x \geq 1$ and $y \geq 1$ values, there is a particular expected score. For example, in all cases where $x = 1$ and $y \geq 1$, the expected score is $1P(⚀)+2P(⚁)+3P(⚂) + 4P(⚃) + 5P(⚄) + 6P(⚅) = 3.5$, where the six possible dice values all occur with probabilities of $P(⚀) = P(⚁) = P(⚂) = P(⚃) = P(⚄) = P(⚅) = 1/6$. In fact, in cases where $y \geq x$, the expected score is given by $3.5x$, since the probability of getting more than y dice with the same value is zero. However, this probability is non-zero in (the more exciting) cases where $y < x$ and the corresponding expected scores are much more difficult to calculate.

The expected scores can be estimated using *Monte Carlo simulation*. More specifically, for each pairing of $x \geq 1$ and $y \geq 1$, you can simply play the game lots of times and calculate the average of your scores. Of course, Matlab can simulate the game far more quickly than you can roll dice! In this exercise, you need to draw a flow chart and write a Matlab function for performing Monte Carlo simulation. The first line of your Matlab function is required to be as follows.

```
function dice(x, y, n, file_name)
```

Here, the arguments `x` and `y` should accept vectors of values for x and y, respectively. For each combination of these x and y values, your function should simulate `n` number of plays of the game and determine the expected score. Your code should write the expected scores into the text file specified by the string argument `file_name`. The text file should use the format exemplified in Figure 4 for the case where `x=1:20` and `y=1:5`. Note that the expected scores provided in Figure 4 are not correct!

The described game can also be played using a pack of playing cards[2]. In this

[1]Each die has six sides, namely ⚀, ⚁, ⚂, ⚃, ⚄ and ⚅. Each side has a value, equal to the number of dots it has. When a die is rolled, one of its sides is randomly selected.

[2]There are 52 playing cards in a pack. Each card belongs to one of the four suits, namely

```
!       y
!x       1       2       3       4       5
 1       3.00    3.00    3.00    3.00    3.00
 2       4.88    6.00    6.00    6.00    6.00
 3       4.57    8.64    9.00    9.00    9.00
 4       2.71   10.48   11.87   12.00   12.00
 5       0.83   10.84   14.57   14.87   15.00
 6       0.06    9.82   16.53   17.86   17.99
 7       0.00    7.52   17.93   20.56   20.94
 8       0.00    4.63   17.90   22.83   23.84
 9       0.00    2.25   16.46   24.56   26.51
10       0.00    0.73   13.74   25.56   29.23
11       0.00    0.14   11.03   25.28   31.24
12       0.00    0.01    7.36   24.39   33.14
13       0.00    0.00    4.34   22.14   33.91
14       0.00    0.00    2.00   18.82   33.66
15       0.00    0.00    0.72   15.19   32.49
16       0.00    0.00    0.16   10.70   30.71
17       0.00    0.00    0.03    7.46   27.67
18       0.00    0.00    0.00    3.96   23.75
19       0.00    0.00    0.00    2.04   19.36
20       0.00    0.00    0.00    0.67   14.66
```

Figure 4: Example of the format required for the output text files. Note that the expected scores provided in this example are not correct!

case, the player chooses the number x of playing cards to take from the top of a shuffled pack. If more than y number of the selected cards belong to the same suit then the player gets a score of zero. Otherwise, the player gets a score equal to the sum of the selected card values, where an ace has a value of 1, a two has a value of 2, ..., a jack has a value of 11, a queen has a value of 12 and a king has a value of 13. For example, in the case where $y = 2$, the $x = 4$ cards 5♡ Q♢ 8♡ A♣ would get a score of 26, since no more than $y = 2$ of the cards belong to the same suit. By contrast, the $x = 6$ cards 5♠ 7♣ 2♢ A♣ J♢ 3♣ would get a score of zero, since the number of clubs is greater than $y = 2$.

As with the dice game, draw a flow chart and write a Matlab function for performing the Monte Carlo simulation of the described card game. The first line of this Matlab function is required to be as follows.

```
function cards(x, y, n, file_name)
```

Here, the arguments of the `cards` function are used in the same way as those of the `dice` function. Again, your code should write the expected scores into a text file having the format exemplified in Figure 4.

Include listings and flow charts for your two Matlab functions in your write-up. Also include the text files obtained by the commands

```
dice(1:20, 1:5, 10000, 'dice.txt')
cards(1:20, 1:5, 10000, 'cards.txt')
```

For both games, identify the optimal value of x for each value of $y \in [1, 5]$.

Here is some advice for completing this exercise:

- Note that when rolling x number of dice, the resultant values will all be independent of each other. However, x number of cards taken from the top of a shuffled pack will *not* be independent of each other, since the x selected cards are guaranteed to have different combinations of suit and value. Your `cards` function should efficiently consider this. You can make sure that your solution is efficient by ensuring that the command `cards(52, 13, 1, 'cards.txt')` takes only a moment to run.

hearts, diamonds, clubs and spades. Each card has one of 13 values, namely ace, two, three, four, five, six, seven, eight, nine, ten, jack, queen and king. No two cards in a pack have the same combination of suit and value. Therefore, there are four cards having each value in a pack and 13 cards belonging to each suit. For example, the four sixes are 6♡, 6♢, 6♣ and 6♠. Similarly, the 13 diamonds are A♢, 2♢, 3♢, 4♢, 5♢, 6♢, 7♢, 8♢, 9♢, 10♢, J♢, Q♢ and K♢. When a pack is shuffled, the order of its 52 cards is randomised.

- You may like to use low values for the argument `n` when testing your functions, in order to get them to complete quickly.
- Draw your flow charts on a computer so that it is easy to move parts around and make changes to them. You should be doing this in parallel with writing your Matlab code, rather than leaving the flow chart to the end. You should find that doing things this way will make it easier for you to break the problem down.
- You will probably find the built-in Matlab functions `rand`, `ceil`, `mode` and `fprintf` to be useful in this exercise.
- Not including argument checking and comments, the `cards` function can be written using only 28 lines of code, without taking any shortcuts to cut this down. Also, a flow chart comprising just 23 items is possible for this game. If you find that your function and flow chart are much bigger than these, then you are probably overcomplicating things.
- Monte Carlo methods have many diverse applications, including weather prediction, the simulation of communication schemes, 3D graphics, economic modelling and artificial intelligence.